Karlsruhe Reports in Informatics 2017,12
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems

Max E. Kramer, Martin Hecker, Simon Greiner, Kaibin Bao, and Kateryna Yurchenko

2017

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems

Max E. Kramer, Martin Hecker, Simon Greiner, Kaibin Bao, and Kateryna Yurchenko

Karlsruhe Institute of Technology
Competence Center for Applied Security Technology (KASTEL), Karlsruhe
firstname.lastname@kit.edu

Abstract. Many software systems have to be designed and developed in a way that guarantees that specific information remains confidential with respect to considered adversaries. Such guarantees depend on the internal information flow inside individual components *and* the system architecture, e.g., the deployment on hardware nodes and properties of their communication links. Therefore, we propose a novel architecture-driven approach for specifying and analyzing the confidentiality of information processed by component-based systems. It includes an architectural analysis that is able to infer leaks of confidential information from abstract architecture models, adversary models, and confidentiality specifications. Our approach supports re-usability of components and specification parts across systems as well as specifications with custom labels, e.g., accessibility of hardware and service interfaces. Additionally, our information flow specifications for components are compositional and supported by tools for non-interference verification on source code level. In two case studies, we show how our specification approach is applied and how the architectural analysis is able to detect information leaks of a system in an early design phase.

1 Introduction and Motivation

In distributed software systems, keeping data confidential while it passes the boundaries of logical components, physical machines, and communication stacks is a major security challenge. To avoid unintended information leaks, the flow of confidential data has to be considered during system design because modifying an existing architecture later to increase the security can be very costly. Therefore, the confidentiality of data has to be addressed in a way that allows early up-front analyses. If components that were already developed and verified can be reused, this can further decrease costs. This requires the analyses to be compositional.

Current approaches analyze *either* inter-component *or* intra-component data flows and focus on specific concerns, such as protocols [13], policies [1], or generated code [18]. However, whether a component-based system fulfills a confidentiality specification can not be evaluated solely with such focused analyses: Many confidentiality leaks are the result of the *combination* of inter- and intra-component data flows and depend on the capabilities of potential adversaries.

Therefore, an integration of sound analysis methods should examine components, their connection and communication as well as adversary capabilities.

In this paper, we propose an approach for specifying and analyzing the confidentiality of information processed by component-based systems based on architectural models and specifications. After modeling the software, hardware, and communication of a system, confidentiality requirements can be added, e.g., the sets of data that shall not interfere or the accessibility of hardware nodes and service interfaces. Potential adversaries and their capabilities to acquire information, e.g., via communication links, can be specified. Our analysis is able to identify vulnerabilities, i.e., potential leaks of confidential information. It is implemented by Prolog inference rules on these models and specifications. In case the analysis finds no leaks, hardware and communication requirements are to be verified in the implemented and configured system as they are not guaranteed automatically.These requirements are made explicit in our model to facilitate later assessment. Software requirements can be transformed to non-interference requirements that can be checked by tools like KeY [26] and JOANA [11] during system verification. In case the analysis finds a leak, the proof-tree is generated to document where and why a system is vulnerable with respect to an adversary. The specification and analysis code as well as all case study models are available online[1].

The approach has four key properties: First, the architecture analysis with generic non-interference rules can directly be applied to *models* with custom confidentiality specification labels. Thus, domain experts can use domain concepts to label system and adversary properties when specifying confidentiality requirements. Both the architecture analysis and the code verification do not need to be adapted to specific systems or adversaries. Second, by inspecting an abstract model, the analysis allows for the identification of architectural designs that cannot meet the confidentiality requirements before the complete system is designed in detail. Third, the analyses are compositional so that results for components or subsystems that were already analyzed can be reused in new systems. Last, the approach scales to complex systems because specifications are only needed for interfaces of system-level components. Specifications for the internal structure of composite components can be added and refined as necessary.

Alltogether, our approach supports developers and architects during the design and analysis of component-based distributed systems so that they can

- specify confidentiality requirements in architectural models for software, hardware, and communication using the graphical editors of the Palladio Bench [24],
- analyze the accessibility of confidential data based on adversary models and architectural models with abstract confidentiality requirements,
- derive source code and annotations for non-interference specifications from models and confidentiality requirements, and
- prove that the manually completed source code complies to the confidentiality requirements using non-interference analyses of KeY [26] and JOANA [11].

[1] https://github.com/KASTEL-SCBS/PCM2Java4KeY
https://github.com/KASTEL-SCBS/PCM2Prolog

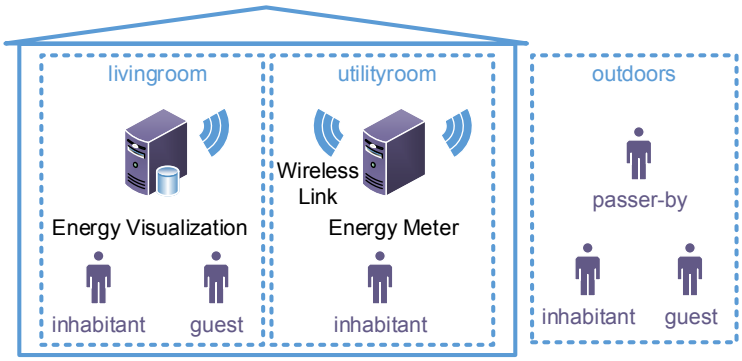

Fig. 1. The example scenario on energy visualization in the household

The paper is structured as follows: In Section 2, we introduce the application scenario of a running example. In Section 3, we explain the concepts of the component-based modeling language that we use and its formal semantics. In Section 4 and 5, we present our extension of this language for the specification of properties used for confidentiality analysis and the new concepts for adversary modeling. In Section 6, we explain the architecture and code analyses based on the modeling language. In Section 7, we present an extension that allows *parameterized* sets of data. In Section 8, we present case studies. In Section 9, we discuss related work and in Section 10, we draw some final conclusions.

2 Exemplary Application Scenario

We briefly introduce a scenario to exemplify the proposed confidentiality specification and analysis. The scenario consists of a household where the energy consumption of the inhabitants is monitored (see Figure 1). Consumption data is acquired by an energy meter and transmitted wirelessly to an energy visualization system. The visualization system stores the consumption data in the internal database and generates consumption graphs for the inhabitants of the household. The energy meter has a wireless consumer interface, which is used by the inhabitant to read the meter values. It also provides additional information for billing purposes, for example the consumer identification. The energy meter is located inside a utility room and the visualization system is in a living room. The locations in Figure 1 are depicted as boxes with dashed lines. Whenever a user is able to gain physical access to a location, a user symbol is placed inside the box. There are three types of users who are interacting with the system: inhabitant, guest and passer-by. Inhabitants have access to all areas but guests have no access to the utility room. Users of type passer-by are not able to enter the house.

3 Formal Architecture Model

In this section, we introduce the architectural model on which our confidentiality specification and analyses are built and we provide a formal definition for the used model elements. We use a subset of the Palladio Component Model (PCM) [24], which is a modeling language for component-based systems that also provides a graphical notation. It is based on the concept of components as composable software units that provide and require services [29]. We introduce a set-theoretic notation for the elements of the architectural model and provide formal semantics for it through the set-theoretic relations and inference rules of our analysis Section 4.

A PCM model for the scenario of Section 2 is shown in Figure 2. For our specification and analysis of confidentiality, we use the following model elements provided by the PCM: *Service*, *Parameter*, *Interface*, *Component*, *Assembly Context*, *Resource Container* and *Linking Resource*. These elements and their relationships are depicted in Figure 8 in the Appendix and described in the following paragraphs.

A *Service* realizes parts of the functionality provided by a system and may have input *parameter*s and a return *Parameter*. In the example, drawEnergyConsumptionGraph() (Figure 2, ①) is a *Service* with no input *Parameter*s (as it uses data provided by the DBMS), but a return *parameter* of type image. In our set-theoretic formalization, the set *Services* contains all *Services* and the set *Parameters* is the set of all input parameters and return parameters. The relation $hasParameter \subseteq Services \times Parameters$ defines the input parameters of a service and the relation $returnParameter \subseteq Services \times Parameters$ defines its return parameter. Both relations are directly extracted from the architectural model of a the system.

An *Interface*, such as DatabaseInterface (Figure 2, ③), groups multiple *Services*. In the set-theoretic formalization, we define *Interfaces* as the set of all interfaces and the relation $hasServices \subseteq Interfaces \times Services$ groups *Services* to *Interfaces*.

A *Component* is a reusable building block of software that is defined independent of its realization. Component-based software engineering is a contract-oriented development approach: every *Component* *guarantees* to provide *Services* via *Interfaces* if the environment provides the *Services* that are explicitly required by the *Component*. In the example scenario, the *Component* Energy Meter (Figure 2, ⑪) provides the *Interface* EnergyMeasurement, which is required by the *Component* Energy Visualization (Figure 2, ⑦). The set-theoretic counterpart of *Components* is the set *Components*. The relation $requires \subseteq Components \times Interfaces$ defines the *Interfaces* a *Component* requires to fulfill its contract. The required *Services* of a *Component* c can be inferred from it. By analogy, $provides \subseteq Components \times Interfaces$ defines the *Interfaces* provided by a *Component*.

To use a *Component* in a concrete system, the *Component* has to be instantiated in an *Assembly Context*. Every *Assembly Context* instantiates exactly one *Component*, but a *Component* can be instantiated in several *Assembly context*s. In the compact representation in Figure 2 the *Assembly Contexts* are represented as boxes surrounding *Components*, e.g. ⑪ for Energy Meter. The set $AssemblyContexts$ is the set of all *Assembly Context* in the model. The relation $componentOf \subseteq$

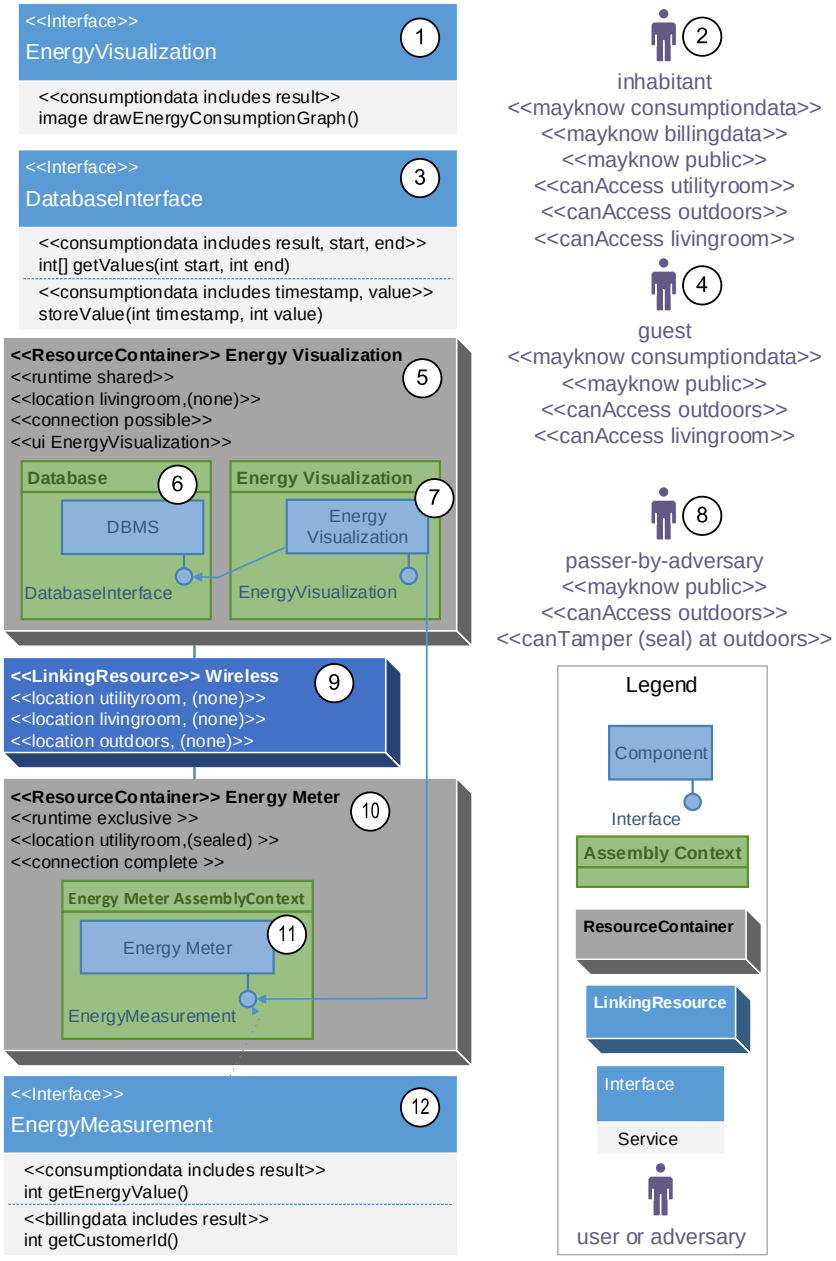

Fig. 2. Specification of the example scenario

$AssemblyContexts \times Components$ defines which *Component* is instantiated using an *Assembly Context*.

A *Resource Container* represents hardware nodes, e.g. servers or embedded devices, on which *Assembly Contexts* are deployed. Every *Assembly Context* can be deployed in at most one *Resource Container* of a resource environment, but a *Resource Container* may host several *Assembly context*s. Our example has two *Resource Containers* on which two (⑤) and one (⑩) *Assembly Context*s are deployed. $ResourceContainers$ is the set of all *Resource Containers*. The relation $runsOn \subseteq AssemblyContexts \times ResourceContainers$ defines the *Resource Container* an *Assembly Context* is deployed on.

For communication, links between *Resource Containers* can be specified using *Linking Resources*. In the scenario, there is only one *Linking Resource* (Figure 2, ⑨) connecting the two *Resource container*s (⑤ and ⑩). All linking resources are represented by the set $LinkingResources$. Which *Resource Containers* are linked by a *Linking Resource* is defined by the relation $connects \subseteq LinkingResources \times ResourceContainers \times ResourceContainers$.

4 Confidentiality Specification

We present a model-based security specification language with a formal basis. To express security-relevant properties for model elements of a component-based system, we extended the concepts of the Palladio Component Model with specification primitives, based on UML-like stereotypes. The specification approach achieves a separation of concerns by specifying confidentiality in two steps:

First, domain-specific labels can be added to resource containers and linking resources to define how systems and components can be accessed by a user of the system, and possible adversaries. Second, input- and output-information of components can be divided into sets to specify information flow properties implied by these sets. All specification possibilities of these two steps directly influence the results of our architecture analysis. The code analysis is only influenced by specifications that are added in the second step.

We use UML-like stereotypes with parameters as a notion for specification in the model. For a model element m being annotated with stereotype s, the relation $hasStereotype(m, s)$ is true. We define relations between system elements and specification primitives as a set-theoretic representation of the specification. In the remainder of this paper, the set-theoretic representation of the model allows us to give formal semantics for the presented security analysis.

When modeling complex systems, the specification language as described here can be verbose. This can be overcome by introducing syntactic sugar into the language, by reusing existing specifications, and by defining implicit default specifications. Experience will show how these abbreviations are introduced the most useful way.

4.1 Domain-Specific Labels

The security requirements for a system usually depend on the domain they are used in. We consider three dimensions of security to be domain-specific: *Locations* on which hardware and wiring is physically installed, *Tamper Protection* mechanisms which are used for securing hardware, and *Datasets* representing types of information processed by the system. It is up to the domain expert to decide which instantiations for these dimensions are useful in a concrete scenario. An overview of the domain-specific labels can be seen in Figure 3.

Symbol	represents	Semantics
Location	Locations in the domain	Each label in the set describes one location. Each location may be a geographic location or a partitioning according to safety levels, specified outside of the modeled system's domain.
Tamperprotection	Methods used for protection against physical tampering	Each label in the set stands for one protection mechanism against physical tampering.
Dataset	Groups of information in the system	Each label stands for a set of information. The content of the set is indirectly defined by input- and output-information. The grouping of information can be performed along user groups or along purpose of the information, like payment information, personal information, or others.

Fig. 3. Domain-specific sets

The example in Figure 2 uses the geographical represenatations

$$Location := \{\mathsf{livingroom}, \mathsf{utilityroom}, \mathsf{outdoors}\}\,.$$

Alternatively, it is possible to define the labels according to levels of protection, like high-security area and similar. A user who has access to the livingroom area also has physical access to a resource container deployed in the location livingroom.

The only restriction for these labels is that it has to be possible to determine unambiguously for every resource container whether it has a certain label or not. An explicit description of the different locations is not part of the modeled system, but is highly recommended as part of the system documentation. Tamper protection mechanisms describe methods used to protect hardware from physical manipulation. Candidates here are for example seals, special screws used for casing and others. The set *Tamperprotection* defines the possibly used anti-tamper mechanisms in the system. Again, the labels can be used to define whatever tamper-protection categorization may be relevant for a given system. It only has to be unambiguously decidable for an expert, whether a resource container is protected by a mechanism described by a certain label or not. We recommend thorough explanations of the anti-tamper methods as part of the documentation.

The example in Figure 2 defines the set of anti-tamper mechanisms as

$$Tamperprotection := \{\mathsf{sealed}, \mathsf{none}\}\,,$$

where none is a default label, which we assume to be always element of *Tamperprotection*. The label sealed describes the usage of a seal on a hardware enclosure. A seal does not necessarily prevent tampering, but it usually allows easy detection of tampering. Adversaries unwilling to risk being caught tampering would refrain from breaking the seal.

Our core concern is the protection of confidentiality of information. We use the concept of *Datasets* to have an abstract representation for categories of information. Again, it is up to the domain expert to decide in which dimensions the categorization of information is performed. In the example, datasets are defined according to the purpose of information by

$$Dataset := \{\mathsf{consumptiondata}, \mathsf{billingdata}, \mathsf{public}\}\,.$$

In other cases, it may be reasonable to define datasets according to stakeholders, who may have an interest in some kind of information, or according to security classifications of information.

The labels defined in *Location*, *Tamperprotection*, and *Dataset* can then be used to provide a system model including security specifications. These specifications are provided in the dimensions of *hardware protection*, *connectivity*, *hardware usage*, *confidentiality of information* and *encryption of communicated information*. Figure 4 shows a list of the stereotypes which can be used for specification in the model. In the following, we will describe their intuitive and semantic meaning and how they are applied in the model.

4.2 Hardware Protection and Accessibility

Hardware can be protected from physical manipulation by placing it in a secure location or by preventing persons from physically tampering with the hardware. Applying the stereotype $\ll location\ l,\ (T) \gg$ specifies that a *Resource Container* or *Linking Resource* is placed on the specified location l and protected against tampering by the specified mechanism T.

Formally, the relation $location \subseteq ResourceContainers \times Location \times Tamperprotection$ is defined as $\{(rc, l, t) \mid hasStereotype(rc, s) \wedge s = \ll location\ l,\ (T) \gg \wedge\ t \in T\}$. The relation $linkLocation \subseteq LinkingResources \times Location \times Tamperprotection$ is defined similarly. In order to specify that at a given location no tamper protection is applied, the special label $\mathsf{none} \in Tamperprotection$ is used.

Especially for a *Linking Resources*, it can be useful to apply this stereotype more than once to express that different tamper protection mechanisms are used at different locations of the same linking resource. A cable, for example, may bridge several locations with different security levels and may be protected differently at each location. In Figure 2, the Energy Meter (10) is specified to be installed in the utility room and it is protected against tampering using a seal. Energy visualization (Figure 2, (5)), on the other hand, does not use any tamper protection

Stereotype	Applicable to	Arguments	Intuitive Semantics
≪*location* l, (T)≫	ResourceContainer, LinkingResource	$l \in Location$, $T \subseteq Tamperprotection$	Combination of location l of the hardware element and the tamper protection T used at this location.
≪*connection* existing≫	ResourceContainer	–	There are connections from the resource container to its environment, which are not explicitly modeled.
≪*connection* possible≫	ResourceContainer	–	If having physical access, it is possible to establish further connections to the resource container.
≪*connection* complete≫	ResourceContainer	–	Even with physical access, there are not further connections to the environment, except those explicitly stated in the model.
≪*runtime* exclusive≫	ResourceContainer	–	All software running on the hardware is explicitly modeled.
≪*runtime* shared≫	ResourceContainer	–	Other software than the one modeled may run on the hardware.
≪*encrypts except* d≫	LinkingResource	$d \in Dataset$	All information communicated over the Linking Resource is encrypted, except information contained in database d.
≪d *includes* P≫	Service	$d \in Dataset$, $P \subseteq Parameters \cup \{/result_{sv}, /call_{sv}\}$	The information encoded in the parameters, return values and calls of sv in P is part of the information abstractly referred to as dataset d.
≪*canAccess* l≫	Adversary	$l \in Location$	The adversary can physically access location l.
≪*canTamper* (T) at l≫	Adversary	$T \subseteq Tamperprotection$, $l \in Location$	The adversary can overcome tamper protection mechanisms T at location l
≪*mayknow* d≫	Adversary	$d \in Dataset$	The adversary may know the information belonging to dataset d

Fig. 4. Overview of confidentiality-relevant specification primitives

mechanisms. The wifi connection (Figure 2, ⑨) covers the living room as well as the utility room. Since a wireless network is used, it is not avoidable that the connection is accessible from the outside, as well. Physical tamper protection, by the nature of a wireless connection, can not be applied. In order to validate the configuration and deployment of a system, a checklist for each resource container and linking resource can easily be generated from the model.

To the knowledge of the authors, there are no tools available that are able to automatically check for an actually installed system whether it satisfies the specification in terms of location and tamper protection mechanisms. Nevertheless, the specification allows an easy generation of a checklist, which states where some hardware has to be installed and how it has to be protected from tampering. It is up to the domain expert, knowing the location specifications and the tamper protection mechanisms to use this checklist for quality assurance purposes.

4.3 Connectivity

Linking Resources represent connections between resource containers in the model. Apart from the connections explicitly modeled, a resource container may provide further possibilities for physical connections. If we assume that there can be no connection that is not modeled, we would have to model all connections for every container in every system, which may not be feasible. Instead, we provide three options to specify connections of resource containers. The stereotype ≪*connection* complete≫ applied to a resource container expresses that only explicitly modeled connections can exist. ≪*connection* possible≫ expresses that the resource container does provide further ports that are currently not connected, for example, USB or Ethernet ports. To express that some of the existing connections are not modeled, the stereotype ≪*connection* existing≫ is used.

The formal representation for connectivity properties is the defined by the relation $furtherConnections \subseteq ResourceContainers \times \{\text{possible}, \text{complete}, \text{existing}\}$ with $\{(rc, c) \mid hasStereotype(rc, s) \wedge s = \ll connection\ \text{c} \gg\}$.

In our scenario, the Energy Meter ⑩ is a closed system that only has a wireless connection. The Energy Visualization ⑤, however, does provide further possibilities for connections, like USB ports and possibly others.

4.4 Hardware Usage

Whether additional software is deployed on a resource container has implications on the confidentiality of information. Similar to the problem of whether or not to completely specify all possible connections of a resource container, it is important whether all deployed software is modeled. The stereotype ≪*runtime* shared≫ expresses that further software might be deployed on a system, as shown in the scenario for the Energy Visualization ⑤. If the system is closed and all software is represented in the model the stereotype ≪*runtime* exclusive≫ is used, as for the Energy Meter in our example ⑩.

The formal representation is given by the relation $sharing \subseteq Resource\text{-}Containers \times \{\text{shared}, \text{exclusive}\}$ with $sharing := \{(rc, sh) \mid hasStereotype(rc, s) \wedge s = \ll runtime \ \ \text{sh} \gg\}$.

4.5 Confidentiality of Information

The goal of our specification and analysis approach is to ensure the confidentiality of information provided to the modeled system as input and provided by the system as output. The stereotype $\ll d \ includes \ P \gg$, with dataset d and parameters P, specifies for a service that all information in the parameters in P is included in the dataset d.

In our scenario, the interface specification for Measurement Acquisition (13) expresses that all information in the input to the service setEnergyValue() is in the dataset consumptiondata, which represents information about the consumption of the household. The specification of the Energy Visualization interface (1) states that all input information influencing the output of the service is in the dataset consumptiondata.

The formal representation of confidentiality properties for services is given by the predicate $includes \subseteq (Parameters \cup \bigcup_{sv}\{/result_{sv}, /call_{sv}\}) \times Dataset$ with $includes := \{(p, d) \mid \exists serv.\ hasStereotype(sv, s) \wedge s = \ll d \ includes \ P \gg \wedge \ p \in P\}$.

The $\ll d \ includes \ P \gg$ stereotype states an information flow requirement for the implementation of a component providing or requiring the annotated interface. It is up to the implementation to ensure that the specified separation of information regarding the datasets is a property of the implementation. A common way to formalize this kind of information flow is non-interference. The semantics we use for formalizing non-interference is discussed in detail in [7], a notion of non-interference specially designed for component-based systems. This non-interference property is compositional and specifications re-usable in different contexts, both properties which a important for component-based systems. Also, the notion of non-interference takes common contracts between components into account, which makes it more precise for our application.

In a nutshell, a component is non-interferent if the environment who can access and read at most system output specified as *low* (i.e. observable), can not gain information about inputs provided to the system and specified as *high*, i.e. confidential. To model the environment, so-called strategies provide input to a component after observing previous outputs sent by the component.

Datasets as abstract information flow specifications allow a black box view on components. Thus, domain experts only require the specification of a component to judge which information may be contained in an output.

4.6 Encryption of Communicated Information

Encryption can be used to ensure the confidentiality of information communicated over insecure channels. But even for encrypted data some information,

like the size of the data or protocol control information (IP addresses, MAC addresses, etc.) cannot be kept confidential. Therefore, the stereotype $\ll encrypts\ except\ d\gg$ can be applied to a *Linking Resource*, to express that all information communicated over it is encrypted except for information in dataset d. If a *Linking Resources* does not have any $\ll encrypts\ except\ d\gg$ annotation, this specifies that no encryption is used at all.

Note that by encryption, the information whether a service was called and whether a service was terminated, can never be hidden. This is because the fact that something is communicated can be observed by observing the presence of communication.

The formal representation of encrypted data is given by the relation $encrypts\text{-}Except \subseteq LinkingResources \times Dataset$ defined by $\{(lr, d) \mid hasStereotype(lr, s) \wedge s = \ll encrypts\ except\ d\gg\} \cup \{(lr, d) \mid d \in Dataset \wedge \neg\exists s, d'.\ hasStereotype(lr, s) \wedge s = \ll encrypts\ except\ d'\gg\}$.

5 Formal Adversary Specification

Our contractual approach towards adversary specification is driven by physical accessibility of adversaries to locations, and abilities an adversary is willing to use. The goal is to support abstract adversary modeling in cooperation with domain experts who can assess which persons have physical access to which parts of the system and can decide which types of adversaries and what abilities have to be considered. This can be done in an iterative process based on previous analysis results or, for example, based on a risk assessment step. The domain expert can exclude particular types of adversaries due to other security measures, e.g., access control to the premises, guards on the premise, or camera surveillance.

By modeling adversaries with certain properties a user of our approach contractually specifies that these properties will hold in the final system. This also applies to the architectural model and confidentiality specification for the system but there is an important difference: for the final system our code level analysis can verify individual components and it would be possible to technically support checks of conformance to the prescribed architecture, e.g. for deployment configurations. But, we have no means to technically verify whether adversaries fulfill the modeled properties.

We model adversaries as entities that are potentially interacting with the system under design. They can represent persons with malicious intentions, but also legitimate users, who are only allowed to use the system in the way prescribed by the architectural model and confidentiality specification. An adversary is represented as an *Adversary* model element and graphically represented as shown in Figure 2 (②, ④, ⑧) and three properties can be specified: First, which domain-specific locations an adversary may gain access to. Second, which actions an adversary is willing to perform in terms of physically tampering with hardware of a system. Finally, which information an adversary is intended to gain access to.

An adversary has physical access to a *Location* l if the stereotype $\ll canAccess\ l\gg$ is applied to it. In Figure 2, the adversary inhabitant (2) has access to all locations in the domain, while passer-by (8) is only able to access the locations outside of the house. It is up to the domain expert to know why an adversary does not have physical access to other locations and by which means this is ensured in order to model the right properties. The stereotype defines the relation

$$locationAccessibleBy := \{(a, l) \mid hasStereotype(a, s) \wedge s = \ll canAccess\ l\gg\}\,.$$

Even if an adversary is, in principle, able to circumvent a given anti-tampering mechanism, he might not be willing to do so. For example, an adversary might be able to pick a lock, and yet refrain from doing so if he has to be afraid of being caught while tampering with it, e.g. due to video surveillance. Therefore, an adversary's willingness of tampering with a given protection mechanism depends on the location. Again, it is up to the domain expert to assess which assumptions according the motivation and ability of tampering should be made. If an adversary is labeled with the stereotype $\ll canTamper\ (T)$ at $l\gg$, he is willing and able to overcome at least one of the tamper protections T at location l. In order to perform tampering, the adversary additionally has to have access to the respective location. The adversary passer-by in the example specification (Figure 2, (8)) has the ability to overcome seals, but can not access to the utility room. The stereotype defines the relation

$$\begin{aligned} &tamperingAbilities := \\ &\{a, l, t \mid hasStereotype(a, s) \wedge s = \ll canTamper\ (T) \text{ at } l\gg \wedge\ t \in T\}\,. \end{aligned}$$

Finally, an adversary might be a legitimate user of a system and therefore obtain knowledge about *some* information processed in the system. In many non-trivial systems everybody may get knowledge about some but not all information. Every ATM user, for example, may know the current system time or information displayed on a welcome screen but not the balance of all accounts. Therefore, we provide the possibility to specify that an adversary may know about information of a dataset d using the stereotype $\ll mayknow\ d\gg$. The relation

$$mayknow := \{(a, d) \mid hasStereotype(a, s) \wedge s = \ll mayknow\ d\gg\}$$

represents this specification formally. In our example, the adversary guest (Figure 2, (4)) may gain knowledge about the consumption data in the system, but he may not get any information about the billing data.

6 Confidentiality Analyses

The confidentiality properties (Section 4, 5) for an architecture model express requirements to be fulfilled by the system. The intra-component information flow requirements as expressed by $\ll d\ includes\ P\gg$ annotations can be checked using

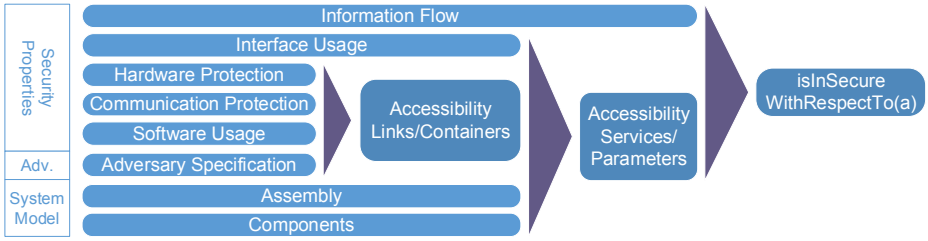

Fig. 5. Work flow of the three-stage architectural access analysis

formal code-level analyses on the actual implementation. Other requirements, such as expressed by $\ll canAccess\ l \gg$, can only be checked informally.

Fulfillment of every single requirement, however, is not sufficient to ensure confidentiality. It also has to be ensured that no violations of confidentiality are inherent to the architecture model. To this end, we propose an automatic access analysis. This analysis reports every unavoidable leak and therefore identifies all faulty architecture models. Only if system external properties do not hold although they are guaranteed, for example because cryptographic keys are not used correctly, the analysis result could be considered a false positive.

6.1 Access Analysis

Under the assumption that

- each component fulfills its information flow specification,
- the protection mechanisms for resource containers and linking resources are properly employed, and location specifications are followed,

the access analysis determines whether the system is *vulnerable* against any of the specified adversaries, i.e., whether they can obtain unintended access to information. For every vulnerability the analysis generates an *explanation* of how a given adversary may exploit it. The explanation guides the architect or domain expert to the model and specification elements that cause the vulnerability.

An adversary who has full physical access to a resource container also has access to any information that is communicated via interfaces available at that container. Similarly, he has access to unencrypted information communicated along linking resources to which he has physical access. We must assume that adversaries are in control of additional software installed on *shared* containers. If, additionally, such software can communicate information to the adversary (by any further connection), the adversary has access to all information available at that container.

The access analysis process is depicted in Figure 5. From the architecture and adversary models and the confidentiality specification the analysis determines which links, containers, and interfaces are accessible to a given adversary. It then checks whether the adversary can access information that is not intended for him by these means, e.g., by circumventing a protection mechanism at a location. The analysis is designed to support extensions to the specification primitives.

Our analysis is explained in terms of *inference rules* that formalize the intuition given for the different confidentiality properties.

Similarly to the relational notation of the architecture model (Section 3), the confidentiality and the adversary specifications are stated in terms of relations between model elements. For example, the annotation $\ll ds\ includes\ P \gg$ states that the relation *includes* holds for all $p \in P$, i.e.: $includes(ds, p)$. In addition to these relations that we directly obtain from the architecture model and the confidentiality- and adversary-specification, we define *inferred* relations (Figure 6), such as $containersFullyAccessibleBy(a, rc)$, which holds whenever an adversary a can—by whatever means—obtain access to all information available at a resource container rc. Ultimately, we are interested in the inferred relation $isInsecureWithRespectTo(a)$, where a is an adversary.

As an example, consider the inference rule for *sharing* and *location*:

$$\frac{\begin{array}{c} containerAccessibleBy(a, rc) \\ sharing(rc, \text{shared}) \qquad furtherConnections(rc, \text{possible}) \end{array}}{containersFullyAccessibleBy(a, rc)}$$

It defines that any information that reaches a resource container rc is fully accessible to an adversary a if a

- can access an resource container physically, i.e., by reaching its location,
- rc is shared, i.e., an adversary can coerce some (unmodeled) program running on rc to access data processed by a modeled component on rc,
- and further connections are possible (to communicate data to the adversary).

The access analysis depicted in Figure 5 is *staged*: relations of one stage are inferred only from relations of preceding stages. Accessibility relations for links and resource containers, for example, are inferred only from the hardware protection, communication protection, software usage, and tampering-properties. Figure 10 in the Appendix shows a selection of the corresponding inference rules.

The analysis answers the question "Is the architecture model vulnerable with regard to the modeled adversaries?". For the example scenario in Figure 1, Figure 2, we start by inspecting the adversary guest. One might be tempted to conclude that there are no vulnerabilities concerning guest, because billingdata is only available from the interface EnergyMeasurement, which is not provided to users by the system. A guest can also access consumptiondata, which is intended. The automated access analysis, however, exposes four vulnerabilities for guest.

The explanation for the first vulnerability is that it is possible for guest to obtain access to billingdata via return values of the service getCustomerId. This service is part of the interface that is used for the link Wireless to connect the resource containers Energy Visualization and Energy Meter. This link is located at livingroom, which is accessible to guest, but billingData is *not* encrypted on the link Wireless. Thus, guest has unintended access to billingData via the link Wireless. The access analysis reports a second, similar vulnerability at location outdoors and a third and fourth vulnerability because the resource container Energy Visualization has a shared run-time and possible connections but no tamper protection.

	Relation r	Arguments	Intuitive Meaning: $r(\ldots)$ holds whenever ..
Links/Containers	$containerAccessibleBy(a, rc, l)$	$a \in Adversaries$ $rc \in ResourceContainers$ $l \in Location$	a can access rc physically by reaching its location l
	$containersFullyAccessibleBy(a, rc)$	$a \in Adversaries$, $rc \in ResourceContainers$	a can – by whatever means – obtain access to all information available at rc
	$linkAccessibleBy(a, link, l)$	$a \in Adversaries$, $link \in LinkingResources$ $l \in Location$	a can access $link$ physically by reaching its location l
	$linksDataAccessibleBy(a, l, ds)$	$a \in Adversaries$, $l \in LinkingResources$ $ds \in Dataset$	a can – by whatever means – obtain access to information from dataset ds transmitted via l
Interface Usage	$providedInterfacesAccessibleTo(a, i)$	$a \in Adversaries$ $i \in Interfaces$	a can directly access interface i provided by the system
	$requiredInterfacesAccessibleTo(a, i)$	$a \in Adversaries$ $i \in Interfaces$	a may offer interface i to the system
Services/ Parameters	$accessibleParameters(a, p)$	$a \in Adversaries$ $p \in Parameters$	a can – by whatever means – obtain access to all information obtainable from the passing of p to some service
	$observableServices(a, s)$	$a \in Adversaries$ $s \in Services$	a can – by whatever means – observe any call to service s
	$parameterAllowedToBeAccessedBy(a, p)$	$a \in Adversaries$ $p \in Parameters$	is a allowed to obtain access to information from parameter p
	$serviceAllowedToBeObservedBy(a, s)$	$a \in Adversaries$ $s \in Services$	is a allowed to observe any call to service s
	$isInsecureWithRespectTo(a)$	$a \in Adversaries$	the system is insecure with respect to a

Fig. 6. Overview of inferred access relations

Given such an explanation, the architect may decide to either require stronger guarantees in the confidentiality specification, to alter the architecture model, or to leave models and specification unchanged, but to document and assess the possible risk associated with the given vulnerability and the corresponding adversary.

To eliminate the first two vulnerabilities in the sample scenario, an obvious modification of the confidentiality specification is to require encrypted communication by labeling Wireless with $\ll$*encrypts except* public$\gg$. This modification imposes new requirements on implementations of the linking resource Wireless.

Alternatively, one may decide to remove any services dealing with billingdata from the interface EnergyMeasurement and to put them into a new interface that is *not* required by the component EnergyVisualization. Implicitly, this forbids the component EnergyVisualization to call any of these services. Hence, no billingdata is transmitted along Wireless, and there is no need to encrypt communication via the Wireless link, at least to achieve confidentiality with respect to guest.

Implementation The inference rules form a stratified [2] (even: hierarchical [4]), allowed—and hence: flounder-free [16]—general logic program. Therefore, we were able to directly implement them in Prolog. The architecture model, adversary specification and confidentiality properties are expressed as Prolog *facts* and the model entities are *atoms*.

For a given adversary a, the analysis result is simply the answer to the query `isInsecureWithRespectTo(a)`. Whenever it succeeds, the system is insecure, and for each violation an explanation is given in form of a proof-tree, which can be generated by a suitable proof-collecting Prolog (meta-)interpreter [23,10,27,30]. The explanation given in the example above is just an informal rendering of the proof tree where the leafs are *facts*. Figure 9 in the Appendix shows the proof tree for the first vulnerability concerning the adversary guest.

6.2 Extensions

The annotations from Section 4 allow a simple and comprehensive modeling of security properties, yet in some applications these may be deemed to unspecific. For example, due to the inference rules, it is implicitly assumed that whenever some adversary can reach some resource containers location, he can also directly access any of the user interfaces provided on that container. Instead, we might want to specify that different interfaces on the same container are protected by different forms of access control (based on, e.g., passwords or security tokens). The access analysis provides optional extensions that support more detailed modeling whenever it is required. Further extensions can be defined easily by extending existing rules and/or replacing simple specification facts by relations inferred from the new specification facts. Investigation of the relations from Figure 6 reveals opportunities for extending the specification and analysis, for example:

Access Control Instead of inferring the relation $providedInterfacesAccessibleTo$ from the system model, we introduce new annotations (and corresponding sets

and relations) that allow formulation of access policies for each interface. For example, for each interface, we allow specification which access domain they belong to, and for each access domain which group of users may access it. The new inference rule will then read:

$$\frac{\begin{array}{c} containerAccessibleBy(a, rc) \\ providedInterfacesOn(rc, i) \qquad isInDomain(i, dom) \\ isMemberOfGroup(a, group) \qquad hasAccessToDomain(group, dom) \end{array}}{providedInterfacesAccessibleTo(a, i)}$$

Detailed Encryption Specification Instead of just specifying for any giving link whether it encrypts or not, one could allow a more detailed modeling of which protocols, ciphers and even implementations are to be used. Together with a knowledge base of (currently known) vulnerabilities of these encryption building blocks, and with reasonable assumptions on each adversaries capability to abuse these, one could replace *encryptsExcept* by a more specific inference rule and a relation $exposesPhsicallyAccessibleDataTo(link, a, dataset)$.

6.3 Code Level Analysis

Annotations $\ll d$ *includes* $P \gg$ specify information flow requirements for parameters and return values of components. In order to check whether a component fulfills these requirement and does not—by accident or on purpose—leak information, we formally verify *non-interference*: Input and output is classified as *low* or *high* and it is required that low output is at most influenced by low input, i.e. low output cannot reveal any information about high input.

For each dataset d, all annotations of the form $\ll d$ *includes* $P \gg$ imply such a classification for our code analyses: In- and output included in d is considered *low*. All other in- and output is considered *high*. A component implementation fulfills its information flow specification if it is non-interfering with respect to all these parameter classifications.

Non-Interference for components is compositional, i.e. non-interferent components can be composed to non-interferent compositions and systems. A thorough account on non-interference in component-based systems is out of scope of this paper, but can be found in [7].

Several tools for language based security can verify non-interference for program code. Two such tools for Java are KeY [26] and JOANA [11]. Both support the specification of non-interference requirements via source code annotations in Java programs, which can be derived from $\ll d$ *includes* $P \gg$ annotations. Similarly, tool independent information flow policies for Java programs in the RIFL specification language [6] can be derived.

Code and architecture analyses do not depend on each other, even if models and code evolve: the architectural analysis can be used if parts of the code specification are outdated and the code analysis can be used if the architectural model is outdated as long as the specification input was corrected according to

code changes. We are currently improving the support for co-evolution of architectural models, specification and code. If the architectural model is outdated, the code can still be verified if the specification input was kept consistent with code changes. Likewise, the architectural analysis could still be used if generated code stubs are not only completed but also modified as long as these changes are not in conflict with the architectural model. Fully automated co-evolution of code, specification and implementation is an unsolved problem in the literature but we already keep parts of JML specifications consistent with Java code [15].

7 Parameterized Datasets

The confidentiality specifications we have described so far assign to each *Parameter* of a service one or more fixed *Datasets*. Any information that was able to interfere with information of the service parameter was considered to be able to interfere with information in the respective *Dataset*. Which component used a service in which context had no influence on these possible interferences.

In complex systems, components often process information from different sources. For example, information provided by different components or information that is only confidential with respect to certain users or adversaries. Components are required to keep such information separate in order to sustain confidentiality of the provided information, even if the information was provided to the same service or service parameter.

Consider a simple cloud storage provider that offers the *Service*s putSelf(file, data) and get(file) to store and retrieve files, as shown in *Interface* FileManagerGUI in Figure 7. It has to be made sure that data that was provided by a given user (i.e.: a client of this *Service*) is only available for this user. More specifically, information in the data service parameter of a call to the putSelf service for a given user must not be retrievable with calls to the get service made by a different user. We cannot express this requirement by including the service parameter data in a single fixed *Dataset*. Instead, we need distinct *Datasets* for each user although the used service parameter is the same.

A user of a service call is usually identified and authenticated by additional parameters which are passed to the called service whenever the service is invoked. Such additional parameters may, for example, contain user-name/password pairs, or an authentication token. For our cloud storage example, we assume that every service of the cloud storage provider has an additional service parameter s, carrying a security token to identify the user. We extend our confidentiality specification approach to support families of separate *Datasets* indexed by corresponding identifiers obtained from values of such service parameters s. In the example, the specification parameter Self, which indexes the *parameterized Dataset* UserData is in each call defined by the service parameter s, as specified by the stereotype $\ll$s *defines* Self$\gg$. Self ranges over {A, B}, giving rise to corresponding *concrete Dataset*s UserData[A] and UserData[B].

When the code of the component-based system, which we partially generate from the models and specification, is verified the relation between service param-

eters that define a specification parameter is as follows: For every specification parameter S such that $\ll s_1$ *defines* S$\gg$, ..., $\ll s_n$ *defines* S$\gg$, a Java method has to be implemented which extracts from values for $(s_1, \ldots, s_n)$ an identifier ranged over by S, or fails if $(s_1, \ldots, s_n)$ do not encode a valid identifier. In our cloud scenario, this extractor may, for example, test if s is a valid authentication token for either identifier A or B.

This extension for parameterized *Dataset*s and specification parameters can be used to define the allowed flow of information for a service of a component prior to the binding of the service parameter. In our example, we specify that at *Interface* FileManagerGUI, the parameterized *Dataset* UserData[Self] includes any *Parameter* (abbreviated: *). In particular, it includes data and the output of *Service* get, with Self ranging over {A, B}, and defined by the service parameter s. This means that an implementation is, in a result for a call to get(name,s'), only allowed to provide data that was obtained in a call to putSelf(file,data,s) if the extractor method returns the same specification parameter value for s and s'. A static verification method for the corresponding notion of non-interference for value-dependent security labels is explained in, e.g. [17].

Restrictions on specification Parameters If only certain values for a specification parameter are allowed in a certain usage context of a service it is possible to specify a binding for this parameter in the architectural specification. This is done by specifying a specification parameter and a set of allowed specification parameter values for an assembly connector between an assembly context of a component that provides the service and an assembly context of a component that requires the service. In the example, the assembly connector for service ObjectStorage required by the component File Manager at User A PC restricts the specification parameter Self to A. This models the requirement that any service call from that component (at User A PC) to the ObjectStorage interface (at Cloud Service) is made with values s such that s encodes the identifier A. Since any valid implementation of component File Manager *is* allowed to handle a request putSelf(file, data, s_b) at it's interface FileManagerGUI by calling the service putSelf(file, data s_b) at interface ObjectStorage, even if s_b encodes B instead of A, this also implies the requirement that putSelf(file, data, s_b) at FileManagerGUI, too, is only ever called with values s such that s encodes the identifier A. Informally: User B never uses A's PC.

Access Analysis With regard to the access analysis, note that wheter a given *Parameter* is included in some *Dataset* now depends on the allocation context a corresponding component is deployed in: In the example, the *Parameter* data of *Interface* FileManagerGUI at User PC A is included in UserData[A], while the same *Parameter* at User PC B is included in UserData[B].Hence, instead of a binary relation *includes* (p, d) on *Parameter*s and (concrete), *Dataset*s, we now have a relation *includes* (p, d, ac) with ac : *Assembly Context*.
Also note that we can, in the example from Figure 7, *not* interpret the specifica-

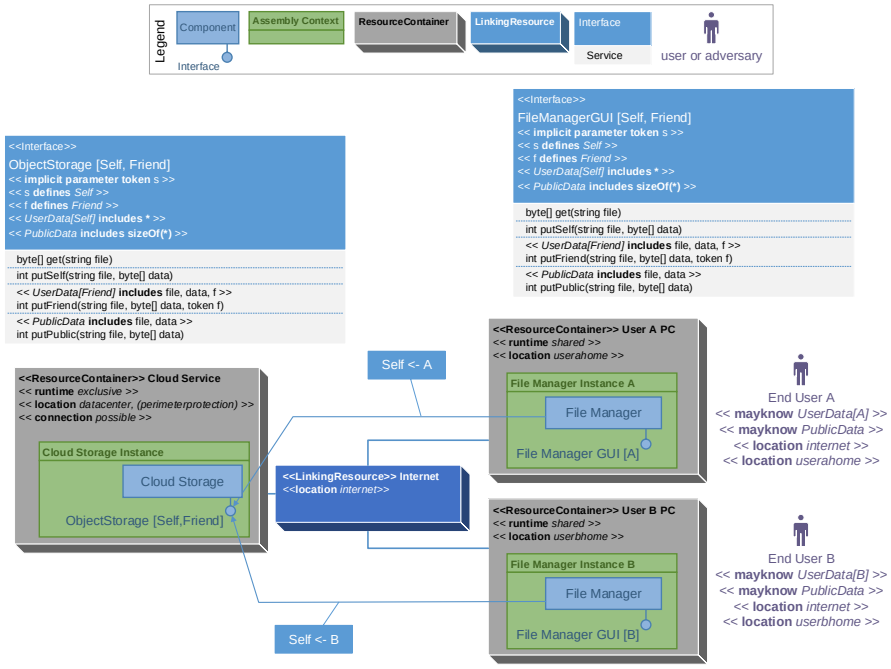

Fig. 7. A simple example scenario using parameterized *Dataset*s

tion to mean that *simultaneously*

$$\begin{aligned} & \mathit{includes}\,(\mathsf{data}, \mathsf{UserData[A]}, \mathsf{Cloud\ Storage\ Instance}) \\ \text{and} \quad & \mathit{includes}\,(\mathsf{data}, \mathsf{UserData[B]}, \mathsf{Cloud\ Storage\ Instance})\ , \end{aligned}$$

since otherwise an attacker who may know UserData[A] (but not: UserData[B]) would be allowed to learn values of, e.g., data (say, by fully accessing the resource container Cloud Service). Instead, we have to analyze two *worlds* separately: one in which

$$\mathit{includes}\,(\mathsf{data}, \mathsf{UserData[A]}, \mathsf{Cloud\ Storage\ Instance})\ ,$$

which has no violation w.r.t. such an attacker, and one in which

$$\mathit{includes}\,(\mathsf{data}, \mathsf{UserData[B]}, \mathsf{Cloud\ Storage\ Instance})\ ,$$

which witnesses the confidentiality violation.

8 Case Studies

We applied our approach to two case studies to evaluate its applicability: In a case study discussing the information-flow security of a travel planer system,

we show that we can express security properties previously discussed in related work in [28]. In a second case study, we modeled a cloud storage scenario. The purpose of this second case study was to explore the kind of security specifications we can provide for a non-trivial system and show how our different levels of security model elements and analysis technique allow to detect errors early in the development of a system.

The models for both case studies can be downloaded online[2].

Travel Planer System The travel planer system of the first case study consists of a travel agency, an airline, a travel planner and a credit card center component. The system offers services for booking flights, confirmation of payment information, setting available flights and others to stakeholders like the airline, a travel agency and customers. An example for a requirement for the confidentiality of information in this system is "The user's credit card data does not flow to the travel agency."

We extended the original specification with explicit adversaries, links and physical access properties. In order to keep our specification as close to the original as possible, we used datasets for each involved party. The original system defines a temporal declassification property: "The credit card data flows to the airline only after explicit confirmation and declassification". We modeled this property, as in IFlow, by adding a declassifying service to the model.

We applied our architectural analysis to the model and verified that there is no architectural leak. As our approach does currently not support declassification annotations in the model, we are not able to generate such source code specifications for KeY or JOANA.

Cloud Storage

As a second case study, we modeled a multi-tenant cloud storage system. The scenario is inspired by a typical medium-scaled deployment of ownCloud[3] or Nextcloud[4].

System Architecture The modeled system consists of components representing databases, logging functionalities, cloud instances and administration tools. In Figure 11, these components are deployed on individual *Resource Containers*. Each of these components are instantiated twice in the system, either for redundancy purposes (database and central logging), or representing copies of the same cloud instance which are run in parallel for performance purposes. We additionally modeled a load balancer which distributes request to the cloud storage system to the two cloud instances. On user level, to model contains components for the cloud client systems (a file manager app and a calendar app), each instantiated

[2] https://github.com/KASTEL-SCBS/Examples4SCBS
[3] owncloud.org
[4] nextcloud.com

for two distinctly different users and a guest user with limited access. These components together provide functionalities for uploading, downloading, and sharing files as well as for managing a calendar with access rights for private-use, friends, or the public.

The components are instantiated on different machines. Two resource containers model virtual machines which are meant to be under the control of a cloud service provider. These resource containers instantiate the central functionality, i.e., the database and central administration tools. Two additional virtual machines, also modeled as *Resource Container*, are meant to be under control of a cloud service administrator. They manage the two cloud instances. Also, a separate *Resource Container* models the virtual machine which runs the load balancer. For each of the three users, a separate *Resource Container* models the machine on which the user runs their client app. The client's machines are connected via the internet to the load balancer, which is connected via an intranet to the machines running the cloud instances, which themselves are connected to the database machines via another intranet.

Security Model Our security model consists of distinct locations representing the home of each user, an outer zone of a computing center, where the load balancer and the cloud instances are located, and an inner zone, where the database is located. In the inner- and the outer zone, special perimeter protections are applied for physically protect the machines from tampering. Apart from the modeled connections, the machines in the inner and outer zone allow for additional connections, if physical access is possible, while for the user machines, which may be personal computers, tablet or mobile phones, we have to assume further connectivity to be existing.

We introduce a parameterized dataset for the users, a distinct dataset for guests: One for logging data, admin data, and general public data. For each interface which provides external access to the overall system, we identified which dataset the input parameters belong to. For each of the outputs, we identified which input datasets the output is influenced by. For all internal interfaces, we decided from a domain point of view, which information is communicated via each service and parameter over the respective interface and added the parameters to respective datasets using our security modeling language.

Attacker Model We modeled several explicit attackers. One attacker represents each user of the system. These attackers have access to the respective location where their client machine is located and they may have access to information which is contained in the dataset for the respective user. A general adversary is modeled by an outsider, who may know only public information and can only access public locations, like the internet. Additional users represent the cloud service provider as well as the cloud administrator. They can access the inner and outer zone, as well as public location and may gain access to logging and admin data, and are willing to tamper with protection mechanisms.

Please note that the purpose of this last attacker type is to show that our analysis actually finds insecurity. An attacker equipped with this power will have

access to user's private information. We do not provide a cloud model which prevents this.

Security Analysis We applied our security analysis, as introduced in Section 6, and identified several causes why we provided a model representing an insecure system.

First, we found that we missed several $\ll d$ *includes* $p \gg$ specifications associating input parameters to datasets. As a result, for example, a user was able to learn input information which was meant to be known only by the user who uploaded it. We identified that this was a mistake in the model. Since the input information was meant to be shared with other users, this insecurity did not actually represent an attack. We fixed this insecurity by adding the respective missing specifications.

Second, we found that basically any attacker was able to read information on users' machines. The cause of this was that we assumed, correctly, in the model that the users' machines had additional existing connectivity, over which user's private information could be read. However, while this is correct, it leads to an unrealistic assumption that any information on a private device is actually knowable by anybody. We therefore made the management decision that the private devices have common security mechanisms installed, e.g., operating system firewalls, sandboxing mechanisms, anti-virus software, and others. While keeping this assumption in mind, we changed the specification of the private devices from existing to possible, expressing that further connectivity to the client's apps is only possible for an attacker with physical access to the device. Note that this assumes that there are no exploitable attack vectors on the client's operating systems.

Another finding was that our system is insecure with respect to the attackers modeling the cloud provider and the administrator, since they can learn user information. Since this was the purpose of these attackers, this finding was not very surprising.

Experience Adding security-relevant specifications to the system model did cause extra effort. For one, we had to decide at modeling time which entities in the system should gain access to which input information on a very detailed level. We also had to decide at modeling time how the components are distributed in a deployment environment at an early stage. The benefit of this was that we were forced to keep security in mind during specification from the very beginning.

Our specification mechanism allowed us to abstract from concrete attack scenarios, where one particular functionality is attacked by one particular attacker. We could reduce security to an abstract location- and stakeholder-focused general consideration of accessibility and who-may-know-what analysis.

Further, the security analysis method as described in Section 6 allowed us to double-check for unwanted mistakes in the model before implementing the system. Without the check, we would have provided incomplete requirements for single components, such that either the components most likely would have been implemented in an insecure way, or quality assurance for single components would

have been incomplete. We also were able to find out that the security concept as modeled contained unrealistic assumptions. We had to make a management decision why we assume user's devices to be more secure than they would realistically be. By being forced to explicitly make an argument why we assume a resource container to be more secure than it actually is, allows us to make this assumption explicit together with a documented argument why we think it is OK to make this unrealistic assumption.

9 Related Work

Various approaches for the design or analysis of distributed systems that process confidential data have been presented in the literature. But to our knowledge, no method for verifying confidentiality requirements both at the design and implementation level according to architectural confidentiality requirements for abstract components was presented so far. A thorough review of model driven development approaches focusing on security properties can be found in [20]. We restrict our presentation here to the most prominent and closest approaches.

UMLsec [13,12], for example, provides analyses for security properties that can be checked formally for the design of individual components. However, the approach does not combine results of implementation and design analyses and only provides an explicit link to source code for role-based access control (RBAC) [19]. UMLsec does not support modeling physical aspects of a system.

Another related approach is IFlow [21][14], which introduced the travel planner system that we used as a case study. The component model of IFlow is similar to that of our approach. IFlow uses application-specific security domains and a specification of allowed flows between security domains to specify the confidentiality of messages (i.e. service calls). As we showed in the case study, our approach allows a similar specification technique using datasets, but also provides additional specification possibilities. IFlow does not support explicit adversaries, specification of physical accessibility properties or links. IFlow, however, supports explicit temporal declassification of information. Our approach does currently not support declassification specification in the model.

Other approaches are restricted to different security concerns, e.g. RBAC for SecureUML [3], or to special domains, e.g. web-services for SECTET [1] or smartcards for SecureMDD [18]. They also lack information flow analyses for code.

The access analysis we propose in this paper is formulated in terms of logical facts and inference rules. It can be seen as an automatic generation of *attack trees* (e.g. [25]) from an architectural system description. Interpreted this way, the adversary capabilities modeled in our approach determine whether the attacks (the tree's leafs) are feasible. Attack tree generation from process algebraic specifications instead of architectural specifications is described in [31]. [22] generate attack graphs from network models, also modeled as logical inference rules. There, the possibility of multi-stage network penetration attacks is analyzed, not the confidentiality of information.

Note that it is not necessary to express security mechanisms and patterns explicitly with our approach. If the effects of a mechanism or pattern are correctly specified, the analysis can consider the resulting information flow. Therefore, it indirectly supports patterns such as complete mediation or minimum exposure.

In [32], the authors discuss a code generation technique, which allows the generation of code stubs for components implemented in Java, enriched with information flow specifications formulated in an extension of the Java Modeling language. In [9], the authors present a code analysis technique for Java Enterprise beans, which can be seen as components implemented in Java. Their approach extends the modeling approach as discussed in this paper with verification tools to ensure that component satisfy the non-interference properties as specified in models. In [8], information flow properties of a cash register system are specified using the non-interference specification mechanisms presented here.

10 Conclusions and Future Work

We presented an integrated method for the specification and analysis of confidentiality in component-based systems. It requires abstract information set specifications for the in- and output of a system and its components as well as accessibility specifications for hardware and communication links. Based on this confidentiality specification and additional adversary models, we presented architecture and code analyses, which can be used to eliminate faulty designs, data leaks, and specification violations.

Our approach provides a consistent way to specify confidentiality at different stages of development and makes it possible to assess confidentiality requirements from the very beginning. Information about the deployment in terms of resource containers, locations, and linking resources as well as user roles and datasets can be specified with custom labels for confidentiality requirements. Based on the judgment of domain experts, it can be specified which users and adversaries are allowed to know information of which datasets and which locations they are allowed to access. Similar, for resource containers and linking resources it can be defined how they can be accessed at specific locations and how they are protected against tampering. By assigning parameters and return values to datasets, we are able to infer which information flow is permissible, thus complementing code-level non-interference analysis methods. At model level, possible violations of confidentiality can be identified in an automated architecture analysis by taking the abilities of adversaries into account. By focusing on the information flow at an abstract level, our analysis method reveals weaknesses at an early design stage but also offers a way to re-use components and code analysis results in a confidentiality-preserving manner.

As future work we will implement support for security properties apart from confidentiality, e.g. integrity. One could also add support for frequently used protection mechanisms, like role-based access control. On the level of components that are deployed on a resource container, it could be useful to have a possibility of modeling sandboxing mechanisms. Furthermore, we plan to evaluate how

much vulnerabilities can be identified with our architectural analysis based on histories of architecture models. We could also take advantage of the fact that the architecture analysis is implemented by a logic program and *infer* adversaries that are able to obtain access to a given dataset by *abductive logic programming* (e.g. [5]) instead of checking for vulnerabilities with respect to a *given* set of adversaries. We plan to investigate the practicability of this approach, e.g. by using a suitable notion of *weakest* adversaries.

References

1. Alam, M., Breu, R., Hafner, M.: Model-driven security engineering for trust management in sectet. Journal of Software 2(1), 47–59 (2007), http://ojs.academypublisher.com/index.php/jsw/article/view/02014759, sECTET
2. Apt, K.R., Blair, H.A., Walker, A.: Foundations of deductive databases and logic programming. chap. Towards a Theory of Declarative Knowledge, pp. 89–148. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1988)
3. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From uml models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. 15(1), 39–91 (Jan 2006), http://doi.acm.org/10.1145/1125808.1125810, secureUML
4. Clark, K.: Negation as Failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Springer US (1978), http://dx.doi.org/10.1007/978-1-4684-3384-5_11
5. Denecker, M., Kakas, A.C.: Abduction in logic programming. In: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I. pp. 402–436. Springer-Verlag, London, UK, UK (2002)
6. Ereth, S., Mantel, H., Perner, M.: Towards a common specification language for information-flow security in rs3 and beyond: Rifl 1.0 - the language. Tech. Rep. TUD-CS-2014-0115, TU Darmstadt (2014)
7. Greiner, S., Grahl, D.: Non-interference with what-declassification in component-based systems. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016. pp. 253–267 (2016)
8. Greiner, S., Herda, M.: Cocome with security. Tech. rep., Karlsruhe Institute of Technology, Faculty of Informatics, Karlsruhe (Apr 2017)
9. Greiner, S., Mohr, M., Beckert, B.: Modular verification of information flow security in component-based systems. In: Cimatti, A., Sirjani, M. (eds.) 15th International Conference on Software Engineering and Formal Methods (SEFM 2017). Lecture Notes in Computer Science, vol. 10469, pp. 300–315. Springer (Sep 2017)
10. Guo, H.F., Ramakrishnan, C., Ramakrishnan, I.: Speculative beats conservative justification. In: Codognet, P. (ed.) Logic Programming, Lecture Notes in Computer Science, vol. 2237, pp. 150–165. Springer Berlin Heidelberg (2001), http://dx.doi.org/10.1007/3-540-45635-X_18
11. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 8(6), 399–422 (Dec 2009)
12. Jürjens, J.: Umlsec: Extending uml for secure systems development. In: Jézéquel, J.M., Hussmann, H., Cook, S. (eds.) «UML»2002 – The Unified Modeling Language, Lecture Notes in Computer Science, vol. 2460, pp. 412–425. Springer Berlin Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45800-X_32

13. Jürjens, J.: Secure systems development with UML. Springer-Verlag, Berlin, Germany (2005), uMLSec
14. Katkalov, K., Fischer, P., Stenzel, K., Reif, W.: Model-Driven Code Generation of Information Flow Secure Systems with IFlow. Technical Report 2012-04, Universität Augsburg (2012), http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/
15. Kramer, M.E., Langhammer, M., Messinger, D., Seifermann, S., Burger, E.: Change-driven consistency for component code, architectural models, and contracts. In: Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering. pp. 21–26. CBSE '15, ACM, New York, NY, USA (2015)
16. Lloyd, J.W., Topor, R.W.: A basis for deductive database systems II. The Journal of Logic Programming 3(1), 55–67 (Apr 1986), http://dx.doi.org/10.1016/0743-1066(86)90004-x
17. Lourenço, L., Caires, L.: Dependent information flow types. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 317–328. POPL '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2676726.2676994
18. Moebius, N., Haneberg, D., Reif, W., Schellhorn, G.: A modeling framework for the development of provably secure e-commerce applications. In: Software Engineering Advances, 2007. ICSEA 2007. International Conference on. pp. 8–8 (2007), secureMDD
19. Montrieux, L., Jürjens, J., Haley, C.B., Yu, Y., Schobbens, P.Y., Toussaint, H.: Tool support for code generation from a umlsec property. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 357–358. ASE '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1858996.1859074
20. Nguyen, P.H., Kramer, M., Klein, J., Traon, Y.L.: An extensive systematic review on the Model-Driven Development of secure systems. Information and Software Technology 68, 62–81 (2015)
21. Ochoa, M., Pape, S., Ruhroth, T., Sprick, B., Stenzel, K., Sudbrock, H.: Report on the RS3 Topic Workshop "Security Properties in Software Engineering". Tech. Rep. 2012-2, Augsburg University (2012)
22. Ou, X., Boyer, W.F., McQueen, M.A.: A Scalable Approach to Attack Graph Generation. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. pp. 336–345. CCS '06, ACM, New York, NY, USA (2006), http://dx.doi.org/10.1145/1180405.1180446
23. Pemmasani, G., Guo, H.F., Dong, Y., Ramakrishnan, C., Ramakrishnan, I.: Online justification for tabled logic programs. In: Kameyama, Y., Stuckey, P. (eds.) Functional and Logic Programming, Lecture Notes in Computer Science, vol. 2998, pp. 24–38. Springer Berlin Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-24754-8_4
24. Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., Kuperberg, M.: The Palladio Component Model. Tech. rep., KIT, Fakultät für Informatik, Karlsruhe (2011), http://digbib.ubka.uni-karlsruhe.de/volltexte/1000022503
25. Salter, C., Saydjari, O.S., Schneier, B., Wallner, J.: Toward a Secure System Engineering Methodolgy. In: Proceedings of the 1998 Workshop on New Security Paradigms. pp. 2–10. NSPW '98, ACM, New York, NY, USA (1998), http://dx.doi.org/10.1145/310889.310900

26. Scheben, C., Schmitt, P.H.: Verification of information flow properties of java programs without approximations. In: FoVeOOS. pp. 232–249 (2011)
27. Specht, G.: Generating Explanation Trees even for Negations in Deductive DataBase Systems. LPE 1993, 8–13 (1993)
28. Stenzel, K., Katkalov, K., Borek, M., Reif, W.: A model-driven approach to noninterference. JoWUA 5(3), 30–43 (2014), http://isyou.info/jowua/papers/jowua-v5n3-3.pdf
29. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York, NY, 2 edn. (2002)
30. Viegas Damásio, C., Analyti, A., Antoniou, G.: Justifications for logic programming. In: Cabalar, P., Son, T. (eds.) Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science, vol. 8148, pp. 530–542. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40564-8_53
31. Vigo, R., Nielson, F., Nielson, H.R.: Automated Generation of Attack Trees. In: Computer Security Foundations Symposium (CSF), 2014 IEEE 27th. pp. 337–350. IEEE (Jul 2014), http://dx.doi.org/10.1109/csf.2014.31
32. Yurchenko, K., Behr, M., Klare, H., Kramer, M., Reussner, R.: Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems. In: MoDeVVa at MoDELS (2017), accepted

Appendix

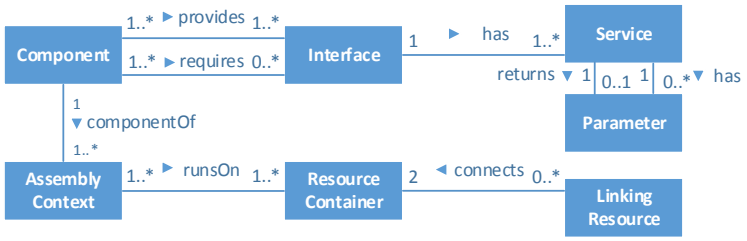

Fig. 8. Entities of architectural models and their relationships

```
isInSecureWithRespectTo(guest)
+- accessibleParameters(guest,return(getCustomerId))
|  +- linksDataAccessibleBy(guest,wireless,billingData)
|  |  +- linkAccessibleBy(guest,wireless,livingRoom)
|  |  |  +- linkLocation(wireless,livingRoom,none)
|  |  |  ‘- locationsAccessibleBy(guest,livingRoom)
|  |  +- linkLocation(wireless,livingRoom,none)
|  |  +- tamperingAbilities(guest,livingRoom,none)
|  |  |  ‘- locationsAccessibleBy(guest,livingRoom)
|  |  ‘- exposesPhysicallyAccessibleDataTo(wireless,guest,billingData)
|  |     ‘- encryptsExcept(wireless,billingData)
|  +- connects(wireless,energyVisualizationRC,energyMeterRC)
|  +- hasService(energyMeasurement,getCustomerId)
|  +- requires(energyVisualization,energyMeasurement)
|  +- parametersOf(getCustomerId,return(getCustomerId))
|  ‘- includes(return(getCustomerId),billingData)
‘- not parameterAllowedToBeAccessedBy(guest,return(getCustomerId))
   +- includes(return(getCustomerId),billingData)
   ‘- not mayknow(guest,billingData)
```

Fig. 9. Proof tree of a vulnerability (excerpt)

$$\frac{location(rc, loc, t) \qquad locationAccessibleBy(a, loc)}{containerAccessibleBy(a, rc, loc)}$$

$$\frac{linkLocation(link, loc, t) \qquad locationAccessibleBy(a, loc)}{linkAccessibleBy(a, link, loc)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, loc) \\ tamperingAbilities(a, loc, t) \qquad location(rc, loc, t)\end{array}}{containersFullyAccessibleBy(a, rc)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, loc) \\ sharing(rc, \text{shared}) \qquad furtherConnections(rc, \text{possible})\end{array}}{containersFullyAccessibleBy(a, rc)}$$

$$\frac{sharing(rc, \text{shared}) \qquad furtherConnections(rc, \text{existing})}{containersFullyAccessibleBy(a, rc)}$$

$$\frac{\begin{array}{c}linkAccessibleBy(a, link, loc) \\ tamperingAbilities(a, loc, t) \qquad location(rc, loc, t) \qquad encryptsExcept(link, ds)\end{array}}{linksDataAccessibleBy(a, link, ds)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, l) \\ providedInterfacesOn(rc, i) \qquad uiInterfaceOn(rc, i)\end{array}}{providedInterfacesAccessibleTo(a, i)}$$

$$\frac{\begin{array}{c}containerAccessibleBy(a, rc, l) \\ requiredInterfacesOn(rc, i) \qquad uiInterfaceOn(rc, i)\end{array}}{requiredInterfacesAccessibleTo(a, i)}$$

$$\frac{\begin{array}{c}providedInterfacesAccessibleTo(a, i) \\ hasServices(i, s) \qquad returnParameter(s, p)\end{array}}{accessibleParameters(a, p)}$$

$$\frac{requiredInterfacesAccessibleTo(a, i) \qquad hasServices(i, s) \qquad hasParameter(s, p)}{accessibleParameters(a, p)}$$

$$\frac{\begin{array}{c}containersFullyAccessibleBy(a, rc) \\ providedInterfacesOn(rc, i) \vee requiredInterfacesOn(rc, i) \\ hasServices(i, s) \qquad returnParameter(s, p) \vee hasParameter(s, p)\end{array}}{accessibleParameters(a, p)}$$

$$\frac{\begin{array}{c}linksDataAccessibleBy(a, link, ds) \qquad connects(link, rcleft, rcright) \\ runsOn(assemblyleft, rcleft) \qquad runsOn(assemblyright, rcright) \\ componentOf(assemblyleft, componentleft) \qquad requires(i, componentleft) \\ systemAssembledTo(assemblyleft, i, assemblyright) \qquad hasServices(i, s) \\ returnParameter(s, p) \vee hasParameter(s, p) \qquad includes(ds, p)\end{array}}{accessibleParameters(a, p)}$$

$$\frac{includes(p, ds) \qquad mayknow(a, ds)}{parameterAllowedToBeAccessedBy(a, p)}$$

$$\frac{accessibleParameters(a, p) \qquad \neg parameterAllowedToBeAccessedBy(a, p)}{isInsecureWithRespectTo(a)}$$

Fig. 10. Access Analysis: selected inference rules. Rules concerning the observation of service *calls* (i.e.: their presence, but not their content) are omitted.

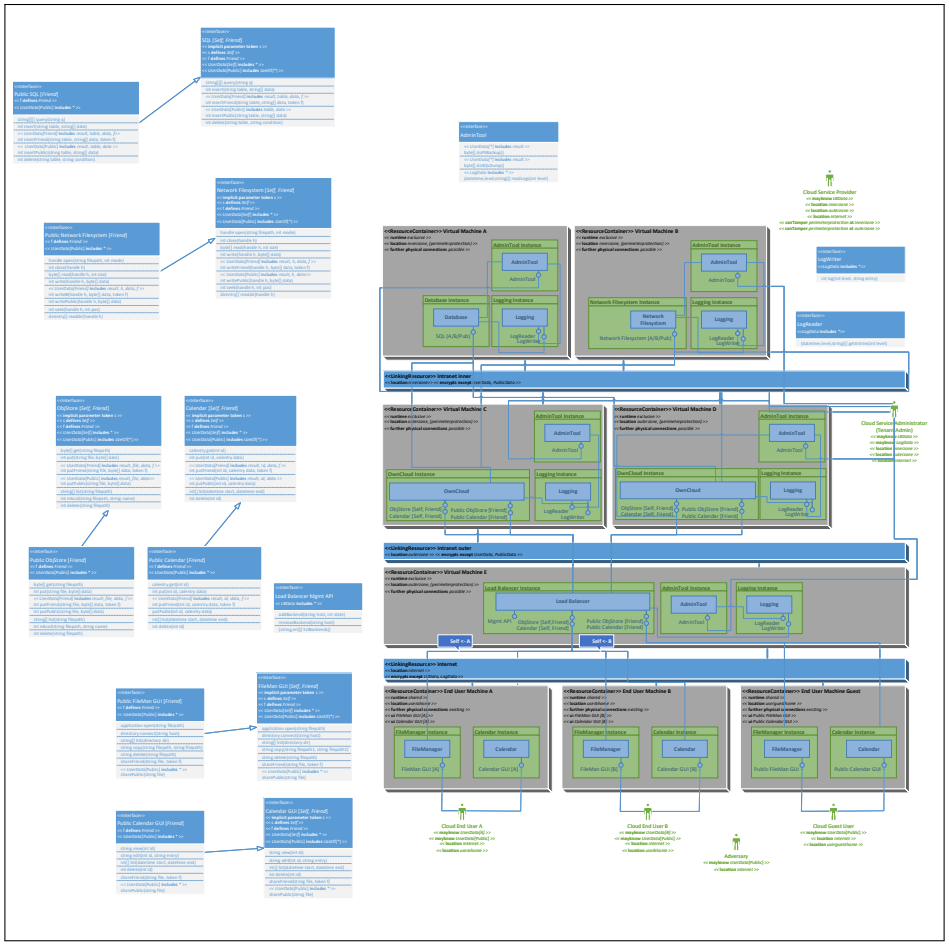

Fig. 11. Case Study: Cloud Storage